A Disk Layout for `dtfs`

Revision: $Id: dtfs-disklayout.lyx,v 1.1 1998/03/30 17:45:59 ceci Exp ceci $

Abstract

This paper shows how data will actually be placed on a physical device within dtfs. The data structures have been designed in a way that should ease the implementation of future enhancements in dtfs without having to alter the on-disk data structures in an incompatible way. Furthermore an overview is given on how versioning and support for multiple logical file systems within one dtfs file system can be implemented using the current on-disk data structures in the future and how dtfs handles checkpoints.

Contents

1 General Disk Layout for `dtfs`

1.1 Design Goals

dtfs' on-disk data structures have been designed with future extensibility in mind. It should be possible to add a number of new features to dtfs without having to alter any of the on-disk data structures. The dtfs on-disk data structures should allow to integrate at least the following enhancements quite easily:

- multiple logical file systems (Up to 255 file systems would be possible without altering the dtfs data structures).
- dtfs Versioning (Up to 64 versions would be possible without altering the dtfs data structures).
- Segment Batching/directory operation log
- Handling of fragments
- Block Sharing
- Files $> 2GB$ (should already work on 64 bit Linux platforms) for the ext2 tradfs.

The design has been made with storage efficiency in mind, too, but the primary focus was on extensibility.

1.2 Explanation of Planned Features

1.2.1 Multiple Logical File Systems

Since dtfs is separated into generic code and code that is specific to one traditional file system making use of the generic dtfs part, it should be possible to place more than one file system (and even different traditional file systems) within one dtfs file system. From a user's point of view this means that a partition containing a dtfs file system could export more than one root directory. Each root directory would then be the starting point for an independent file system tree, a *logical file system*.

1.2.2 dtfs Versioning

Probably more useful than placing multiple filesystem in one dtfs filesystem is the ability to do dtfs versioning. This feature would allow the definition of *versions* of a logical file system at any point in time. Such a version could then be turned read-only (*snapshot*) or could again be used for read/write access (*clone*) resulting in two different versions of a file system existing from that point in time onwards.

Having the ability to define read-only snapshots of a file system while users can still access the file system in read/write mode is quite convenient for backup purposes. A backup is meant to represent the status of a file system at a certain point of time.[1] In reality the problem with backups is that it takes a considerable

[1] This is required for consistency reasons.

amount of time (minutes to hours) to complete the backup since the speed of current backup media is rather limited.

If the file system is accessible in read/write mode by users while the backup is in progress, the state of the backup being generated is not a well-defined one anymore since a user might alter files while the backup is in progress and the actual file version being saved on the backup medium is subject to a race condition between the user's process altering the file and the backup process reading it in and copying it to the backup medium.

In order to avoid that problem, access to a file system is turned off (or switched to read-only) for ordinary users to the file system during backup time. However, this may result in some important services being temporarily unavailable while the backup is taken. — This is one reason why backups are often performed at a time when only very little user activity can be expected. In many environments such a "denial of service approach" can still be unacceptable.

A dtfs implementation with versioning support could circumvent that problem by defining a read-only version of the file system to be backed up at a certain point of time t. Creating such a snapshot should not last much longer than creating any other dtfs checkpoint, so the delay should be within a limit that is expected for normal file system operations. This snapshot could then be used as a starting point for a backup that is guaranteed to match the filesystem status as it was at the time t, while users can still continue to access the filesystem in read/write mode while the backup is created..

1.2.3 Segment Batching/directory operations log

As outlined in section 4 on page 14, dtfs does not use any techniques that allow sophisticated roll-forward algorithms right now. (In order to compensate for that, dtfs introduces more lightweight checkpoint data structures that allow checkpoints to be written more frequently as described in section 4.)

The LFS implementation in 4.4BSD uses a technique called "segment batching" to support roll-forwards, while the Sprite LFS maintains a special directory operation log.

Since the on-disk data structures of dtfs are designed in a way that using a unified area for segment description and directory operation logging, adding a directory operation log should be quite straightforward. Segment batching could be added to dtfs, too, since it does not require any additional on-disk data structures.

1.2.4 Handling Of Fragments

Different strategies can be used in order to address the problem of efficient handling of small files. One approach is to de-couple the allocation of disk space from a block-oriented strategy at all.[2] Since the first implementation of dtfs will be based on ext2 data structures, it seems natural to keep the block-oriented allocation strategy that is used by ext2 in oder to keep things simple.

[2] This approach has been taken by the designers of the Spiralog file system, for example.

ext2 is already prepared for the addition of fragments. Fragments are sub-block units of disk space allocation that help to reduce the waste of disk space caused by even small files allocating a full disk block.

Fragments can be supported by dtfs by adding additional entries in the segment description area that describe the contents of a fragment block (i.e: which fragment belongs to which file).

1.2.5 Block Sharing

Disk space could also be preserved by applying a similar techniques as used by [MERGEMEM] to disk blocks: When file A as well as file B contain a block with identical data in it, it would be possible to allow the meta-data of both files to point to the same physical block and to employ a copy-on-write strategy if such a block should be changed in one of the two files.

A user-space program running at low priority could scan the files on the file system for blocks that can be shared and present suggestions to the kernel implementation of dtfs concerning such merges of blocks.

Again, this could be covered by allowing more than one entry in the segment description area of each partial segment.

2 Special dtfs Disk Blocks

2.1 Overview of the dtfs Disk Layout

2.1.1 Whole Disk Layout

Figure 1 shows an overview of the dtfs disk layout. The reserved area at the beginning of the device holds the dtfs super block (at a fixed location), the traditional filesystems' super block(s) and the checkpoint block(s). The super block and the traditional filesystems' super block(s) are replicated in regular intervals. In oder to compensate for that, the respective partial segment will be a bit smaller than the other partial segments.

Furthermore all the superblocks and the checkpoint blocks will be replicated in a reserved space at the end of the device.[3]

2.1.2 Metadata References

Figure 2 shows an overview of how the different kinds of metadata information are linked together. Starting from the dtfs superblock, the superblock of the respective traditional file system and the checkpoints can be found. There are two different checkpoints for each logical file system and each checkpoint is replicated once at the end of the device resulting in a total of four checkpoint blocks per logical file system.

[3]Should the checkpoint blocks be replicated more frequently too? – But this would probably gain not too much since we have a single point of failure at the partial segment header of the segment containing the checkpoint anyway.

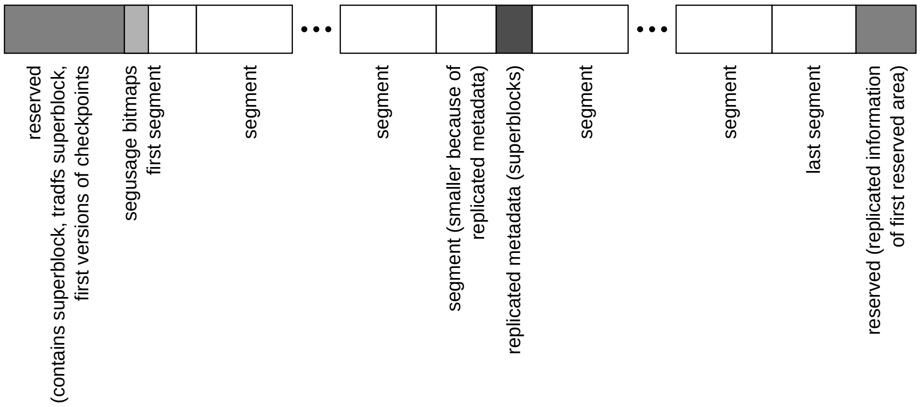

Figure 1: dtfs disk layout overview. Device block numbers are increasing from left to right.

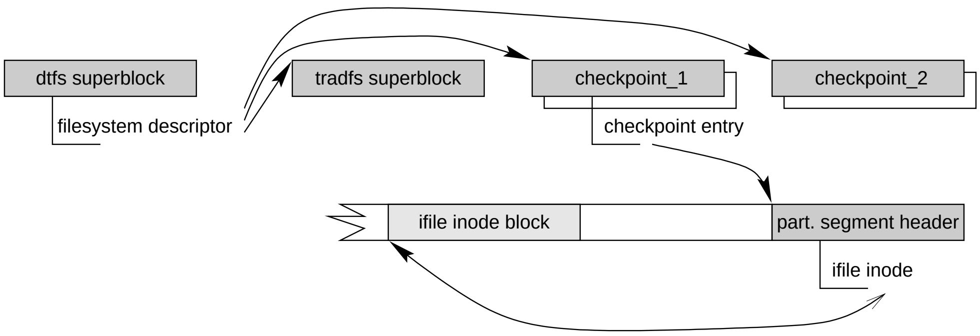

Figure 2: An overview of dtfs metadata referencing. Device block numbers increase from left to right and from top to bottom.

Furthermore there are two segment usage bitmap areas, one for each par of checkpoints. They are located before the beginning of the first segment with the segment usage bitmap for the first pair of checkpoints coming first.

The current checkpoint block pair references a partial segment that contains the actual checkpoint. The header of the partial segment in turn lists the physical block number of the data block containing the ifile inode. This allows all the necessary information for mounting the file system to be obtained.

2.1.3 Partial Segment

The partial segment is the unit of disk space allocation and the unit of cleaning. A partial segment can either be marked as clean or as occupied by the log. Furthermore, a partial segment can be marked as having a checkpoint in it, as being locked (i.e.: no cleaning is allowed to take place) or as being bad (it will not be used for storing log information in it). However, the partial segment does not have any further special semantic for dtfs.

Figure 3 shows the layout of a partial segment. The partial segment header is placed last in the partial segment. This should ease the implementation of a transaction-like semantic for the partial segment write: When all the blocks in a partial segment are written in ascending order, writing the partial segment's header commits the transaction of writing out a partial segment.

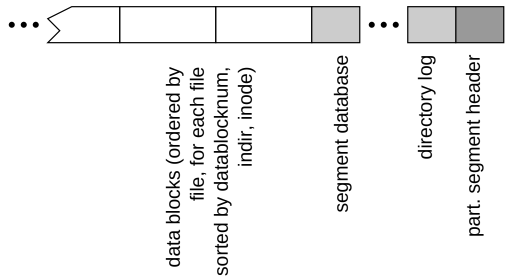

Figure 3: Schematic layout of a partial segment in dtfs

The data blocks in a partial segment are clustered together file-by-file. A file's data blocks (ordered by their blocknumber within the file) are *immediately followed* by single indirect blocks, double indirect blocks and triple indirect blocks (in case of the ext2 filesysetm being used as tradfs on top of dtfs). The inodes are located in the ifile directly (see section3 on that topic). The blocks belonging to the inode file will always be placed last.

2.2 Description of the dtfs Metadata Blocks

The current design for dtfs uses the following metadata blocks in order to define its disk layout:[4]

`dtfs_super_block` A super block to define general key items of a dtfs filesystem.

`dtfs_checkpoint_block` Blocks that hold information about checkpoints and versions of a logical file system.

`dtfs_psegment_block` A block that describes the beginning (or actually the end) of each partial segment.

`dtfs_segdatabase_block` Blocks for storing directory log and segment content description information.

[4] Apart from the blocks listed below, dtfs also requires a superblock for the traditional file system for each logical file system in the dtfs file system.

2.2.1 dtfs_super_block

This data structure is placed on a well-defined block of the device and will be replicated several times on the device. It contains vital information about the filesystem layout on the device. Furthermore, it contains information about the logical file systems found in the respective dtfs filesystem. Currently, up to eight filesystem descriptors can be stored directly in the super block.[5] For further extensions, it is also allowed that the superblock is preceded by a number of blocks holding filesystem information only should the number of logical file systems exceed eight.

The super block also stores the size of a filesystem descriptor. This makes it possible to add extensions to the filesystem descriptors in later versions without the risk of loosing compatibility.

The super block also contains five unused 4 byte entries at the moment that could be used for future enhancements.

It is also expected that a dtfs implementation replicates the filesystem descriptor blocks and the traditional filesystems' super blocks whenever it replicates the dtfs superblock.

2.2.2 dtfs_checkpoint_block

This data structure holds all information about checkpoints for a special logical file system. Each logical file system has two different checkpoint areas that are used by dtfs alternately. A timestamp within each checkpoint block indicates which one is the current. Every checkpoint is replicated once at the end of the device to avoid single point of failures.

A checkpoint block contains a list of checkpoint entries. Currently up to 42 checkpoint entries can be in a checkpoint block. Future dtfs implementations are expected to add additional blocks that contain a list of checkpoint entries only, immediately after the respective checkpoint block.

Checkpoint entries are described later in chapter 2.3.

2.2.3 dtfs_psegment_block

Every partial segment is ended with a dtfs_psegment_block. This block has pointers to the previous and next segment in the log as well as a time stamp. Furthermore it holds information for the cleaner (segment database and directory information log sizes, flags, segment size, etc...).

The partial segment block also contains the start of the segment description holding the segment database and directory log entries for the current segment. Up to 29 of these entries can currently be stored in the partial segment header. If there are more than 29 of these entries to be stored, additional blocks holding the rest of the partial segment information will immediately precede the partial segment block.

[5] filesystem descriptors will be explained in section 2.3.

If the segment is a checkpoint it also contains the physical address of the block that holds the "special inodes", i.e. at least the ifile inode.[6] Furthermore there is also room for two 4 byte checksums in the partial segment header.

A flags field is also present in the partial segment header. The flags indicate whether a particular segment

- is clean
- is locked (i.e: must not be cleaned)
- has bad blocks (segment will never be used)
- contains a checkpoint.

Two entries reserved for 4 byte checksums can also be found in the partial segment header.

Genreally, the checksums should not be needed. However, for the implementation of 4.4BSD LFS checksums were added to the partial segment block. The reason for this design decision was that in BSD the order in which block writes in multiblock write operations are actually performed, cannot be determined. — According to the Linux Kernel Hacker's Guide, Linux uses an implementation of the elevator algorithm to re-order I/O requests. This approach guarantees that Linux will do block writes in ascending block number order. However, reserved entries for checksums are still there in the dtfs segment header since it might be possible that either a re-ordering of writes is again done in the harddisk controller and to avoid the necessity of an overhaul of the partial segment header structure should Linux' write policy be changed.

The first implementation of dtfs will not use these checksum areas and rely on the elevator algorithm to guarantee that the partial segment header is actually written out last.

2.2.4 dtfs_segdatabase_block

One or more blocks of this type may precede a partial segment block if the directory log and the segment description do not fit completely into the partial segment block itself. Only the first segment segment database block in a row may have unused entries. The total number of used segment database entries and the directory log entries in all the segment database blocks and the partial segment block itself must match the respective numbers specified in the partial segment block.

Furthermore, the directory information log is always expected to be located before the segment database which has to be placed immediately before the partial segment block.

[6]The ifile inode is supposted to be the first inode in this block.

2.3 Additional Data Structures

The metadata blocks described in the previous section will also make use of a bunch of other data structures:

dtfs_filesystem_descriptor A data structure used within the super block that holds all the required information for a logical file system.

dtfs_checkpoint_entry Used within the checkpoint blocks to describe a certain checkpoint for a logical file system.

dtfs_dirlog_entry A 16 byte structure describing an entry in the directory operation log.

dtfs_blkdesc_entry Another 16 byte structure that contains the description of a certain block in a partial segment.

dtfs_segdatabase_entry Either a dtfs_dirlog_entry or a dtfs_blkdesc_entry.

2.3.1 dtfs_filesystem_descriptor

A dtfs superblock holds a number of filesystem descriptors. Every filesystem descriptor can be used to describe one logical file system within the dtfs file system.[7] The filesystem descriptor contains the address of the checkpoint regions for the respective filesystem, the inode ids for some crucial inodes (ifile, root inode, segusage inode[8]) and a flags field that is currently used for marking the filesystem as "unmounted cleanly" and as "readonly"[9].

2.3.2 dtfs_checkpoint_entry

A checkpoint block is made up of several checkpoint entries. A checkpoint entry holds the timestamp at which the checkpoint was made and the address of the partial segment containing the checkpoint. A flags field indicates whether the checkpoint is current, a version (r/w–>clone, r/o–>snapshot) or read-only.

2.3.3 dtfs_dirlog_entry

An entry describing a directory operations log. It consists of an operation type indicator, the inode of the parent directory and the file (or directory) affected. Furthermore it contains the number of the data block of the parent directory that has been changed by the directory operation.

The directory operation log is expected to be in the first segment containing one of the directory modifying file operations.

[7] The first implementation of dtfs will only use the first of these entries since multiple logical file systems will not be supported in the first dtfs implementation.

[8] The segusage file would be shared by all logical file systems or just be present in one logical file system, however.

[9] Altough this will probably be handled by the traditional file system.

Contrary to Sprite LFS, dtfs does not store the name of the child file or directory affected by the change since this would use additional amount of storage and complicate parsing of the directory log/segment database. Furthermore, the benefits from storing the filename of the child in the directory operation log are questionable anyway.

Please note that dirlog entries are defined by dtfs, but not used, since dtfs uses lightweight checkpoints instead of directory operation logs. (See section 4 for more information on that).

2.3.4 dtfs_blkdesc_entry

A data structure describing the contents of a certain block in the current partial segment. It contains a block type identifier (inode, single-, double-, triple indirect block, data block). Furthermore the id and version of the filesystem the block belongs to is also stored here.[10] The inode id of the owner and the block offset of the data block within the file are stored here too. In addition, the file version information is also kept in that data structure.

2.3.5 dtfs_segdatabase_entry

Actually a union of directory log entry and block description entry. Whether an entry is either a block description or a directory log entry can still be determined by the type field.

3 Filesystem-Mapped Metadata Information

3.1 Overview

dtfs tries to map as many metadata information as possible into regular files. This helps to avoid having to deal with separate metada information types for the traditional file system used together with dtfs and for dtfs itself. dtfs gets along with only using the metadata structures provided by the respective traditional file system Therefore an improvement to the filesystem meta data structures in the traditional file system will also be available for dtfs metadata handling as well.

The following dtfs data structures are mapped into regular files:

1. The .inode file: All inodes are mapped into a single file called .inode

2. The .iusage file: This file contains information about the inodes+ allocation data. It is used to locate a free inode quickly, should the need arise.

[10] This additional information should greatly ease cleaning when multiple filesystems/versions are present.

3. The .atime file: It contains the current version number and the time of the last access for each file. This information is not stored into the inode itself directly because

   (a) Every read access to a file needs to update the atime entry of the respective inode. Writing the entire inode structure would result in an unnecessary overhead that can be avoided by keeping the inode data and the time of the last access to the file separated.

   (b) The version number of the file as used by a log-structured file system cannot be found in the inode structure of a traditional file system. In order to avoid portability problems, it is a good idea not to ad the atime filed to the inode of the traditional file system (since it cannot be expected to conveniently fit into the inode structure that is already used by the traditional file system.)

3.2 The .inode File

This file contains all the inodes of the traditional file system with the exception of the inode for the .inode file itself.[11] In order to allow quick access to a special inode, dtfs relies on the metadata structures of the traditional file system, since inode accesses are finally translated into simple file accesses, too.

3.3 The .iusage File

3.3.1 General Purpose Of This File

This file contains a bitmap and some metadata information (called "meta-bitmap") indicating available inodes, that are not in use by now. The Sprite LFS implementation uses a linear bitmap in order to mark unused inodes. Locating a free inode is done by a linear search through that bitmap starting from the bit representing the directory inode holding the parent directory of the file to be created. The advantage of this approach is that inodes for files located in the same director are kept close together resulting in good locality.

3.3.2 Inode Locality vs. Finding a Free Inode Fast

The problem with this approach is that it involves a linear scan through the inode bitmap. [SELTZER] investigated that problem and has found out that the Sprite LFS has to search an average of 94 bitmap entries (worst case is approx. 120 entries) before it finds a new inode that is available. Therefore this approach is not very efficient.

In order to overcome these problems, the BSD LFS implementation uses a list of free inodes that are available for allocation. However, such an approach

[11] However, the current ext2 implementation of dtfs reserves space for the ifile inode in the ifile itself, but does not use it.

makes it harder to get all the inodes of files belonging to a specific directory clustered near by the directory inode itself.

dtfs uses another allocation scheme that should provide both an acceptable inode location performance and good locality for the inodes of all files within a directory.

3.3.3 dtfs Inode Meta Bitmaps

From the data points found in [SELTZER] concerning the Sprite LFS free inode location performance, it can be concluded that a free inode can be found within the first 120 inodes following the parent directory inode in almost all the cases. So a data structure that speeds up the scanning for a free inode in the vicinity of another given inode has been added to dtfs: Every filesystem block belonging to the .iusage file does contain a meta bitmap. Every bit in the meta bitmap represents a 4 byte word in the inode allocation bitmap. The bit is set if all the 32 bit of the inode allocation bitmap it represents are set too, i.e: if all inodes represented by this 4 byte word are already in use.

Since the meta-bitmap is stored directly in the filesystem block it describes, no additional disk reads are required in order to retrieve the meta bitmap. The only drawback of this approach is that the layout of this data structure is therefore dependent on the logical file block size used. Table 1 shows theactual layout of a .iusage file block for all the logical block sizes that are currently supported by dtfs.

log. file block size	size of meta bitmap	# inodes represented by block
1kb	8 (last 8 bits unused)	248 * 32
2kb	16 (last 16 bits unused)	496 * 32
4kb	32 (last 32 bits unused)	992 * 32

Table 1: inode meta bitmaps for various logical file block sizes. (Sizes in 4byte words)

However, extending this data structure to a tree-like one has been taken into consideration, but given the fact that a free inode can be found within the next 120 inodes starting from any given directory inode, the benefits of such a data structure would be questionable compared to the added complexity.

By using meta-bitmaps for inode allocation a free inode should be found by just looking at two 4 byte words[12] (one in the meta-bitmap and one in the actual allocation bitmap) in almost any case.

3.4 The .atime File

This file contains to 4byte entries for each inode representing the last time the file was accessed and the LFS version number of the file that is required by the cleaner.

[12] Contrary to the 12 4 byte scan required by the Sprite LFS.

4 Checkpoints

4.1 Filesystem Checkpoints

The first implementation of dtfs does not make use of segment batching or a directory operations log. In order to cope with operations that require more than one file system modification (like the creation of a file), dtfs writes out checkpoints. Checkpoints mark a consistent state of the filesystem at a certain point of time. So if a checkpoint is use to re-mount a file system, the file system can be considered to be in a consistent state.

As a consequence of this approach, however, no roll-forward algorithm can be employed in order to recover as much filesystem modifications as possibly after a crash. In order to compensate for that, dtfs has to write out checkpoints as often as possible.

The problem with checkpoints is that they might be rather costly to write thus degrading overall filesystem performance. Furthermore writing checkpoint out rather often might result in increased cleaner activity since the cleaner has to free the space occupied by checkpoint-blocks that are no longer live.

In order to deal with that problem dtfs employs different checkpoints. Every kind of checkpoint defines a consistent state of the file system at a certain point of time that allows to remount thefile system after a crash. However, the time required to rebuild all the internal dtfs data structures might differ:

4.2 Filesystem Checkpoint Types

dtfs provides the following different checkpoint types:

4.2.1 Unmount Checkpoint

An unmount checkpoint is written only when a filesystem mounted for read/write access is being unmounted. Writing an unmount checkpoint requires the fillowing steps:

1. All dirty data and metadata blocks are written out
2. A checkpoint block holding the ifile inode (and maybe some hints for the cleaner) is created.
3. A partial segment header with the SEG_HAS_CHECKPOINT flag set is created.
4. The current segment usage block is updated.
5. The current checkpoint areas are updated.

All the dtfs super blocks are updated and re-written.[13]

If a file system is unmounted cleanly, there is always an unmount checkpoint present that describes the state of the file system at the time it was unmounted.

[13] Or should it be sufficient to just update the first super block. If that super block copy cannot be read in, it means that the file system has not been unmounted cleanly.

4.2.2 Major Checkpoint

A major checkpoint is written out by dtfs every now and then (implementation dependend). Writing a major checkpoint involves the same steps as writing out an unmount checkpoint with the exception of step 6 (writing out the super blocks).

4.2.3 Minor Checkpoint

Writing out a minor checkpoint involves the steps 1–3 of writing out an unmount checkpoint, i.e: writing out a minor checkpoint only involves writes to the tail of the log, so no (big) disk head movements should be required to write out a minor checkpoint.

4.2.4 NFS Checkpoint (not implemented in first dtfs version)

Write operations in a log-structured filesystem get particularly inefficient when files are written in many small blocks and a sync request is issued after each small block that has been written (as it is the case for NFS writes with the default, small block sizes). In order to optimize such access patterns, it would be possible to design another checkpoint:

Such writes could be handled by grouping two segments that are close to each other on the physical device to a segment pair. Each segment of the segment pair has an additional checkpoint area holding the segment description information of the current partial segment of both segments in the segment pair.

NFS block writes alternately go to the partial segments in the two segments of the segment pair. The segment description information are updated alternately, too. (Writing that segment description information serves as a commit in that case).

However, this strategy would also avoid having to write the filesystem meta data for the affected files again and again since the necessary metadata information could be reconstructed from the segment description in case of a system crash.[14]

4.3 Log Checkpoints

A log checkpoint is just another term for partial segment header. Since a segment is the unit of cleaning, a partial segment must not be allowed to extend beyond the end of a segment. Therefore the log may decide to add an additional partial segment header just to end a partial segment at a segment boundary.

4.4 Recovering From A Crash

If a dtfs file system has been unmounted cleanly, there will alway be an unmount checkpoint present allowing the efficient re-construction of all internal

[14] Please note that the NFS checkpoint is just a proposal and it will not be implemented in dtfs in the next future.

dtfs data structures that are necessary for operation. If a file system has not been unmounted cleanly, dtfs starts up by reading in the last unmount checkpoint listed in the checkpoint areas. From that starting point a forward scan of the log takes place. The forward scan continues until another partial segment header is found that has the SEG_HAS_CHECKPOINT flag set.

By updating the segment usage bitmap according to the segments that have been traversed since the original checkpoint segment and by updating the checkpoint areas this newly found checkpoint is turned into a major checkpoint. The algorithm starts again from that point to search for the next partial segment header with a checkpoint. If none is found in the remaining log, the log is truncated at the last partial segment with a checkpoint in it.

However, if there is no pointer to a valid checkpoint in the checkpoint areas to be found, only a sequential scan of the whole device can be used to re-construct the log. But this case should only happen in the case of a bug in the dtfs kernel code or because of a disk medium failure.

5 ext2 Personality-Specific Features of dtfs

5.1 The 2GB File-Size Limitation

5.1.1 Why There Is A 2 GB Limitation At All

Currently Linux can handle only files with a maximum size of 2 GB. This limitation is caused by two reasons:

1. The Linux VFS layer cannot deal with file offsets $> 2^{31}$ bytes on 32 bit platforms. [15]
2. The ext2 on-disk inode data structure does not allow to store file sizes that do not fit into a 32 bit word which would result in a maximum file size of 4 GB. Due to other constraints, the current ext2 implementation in the Linux kernel deliberately limits the file size to 2 GB. However, the indirect block structure of ext2 would allow file sizes of up to 4 TB on a filesystem with a block size of 4 kb.

While it is impossible to overcome the first reason without major changes to the Linux VFS layer, it is possible to address the second issue in a filesystem implementation.

5.1.2 Avoiding the 2 GB ext2 Limitation in dtfs

The current dtfs traditional file system implementation is based on ext2 and it is desirable to keep the dtfs tradfs implementation close to the ext2 implementation in order to be able to incorporate future ext2 enhancements into dtfs effortlessly. So using one of the currently unused entries in the ext2 inode structure to

[15] However, at least the Linux/Alpha 2.0.33 kernel together with Jay Estabrookt's axp-diffs does not seem to have that problem.

represent file sizes $> 2GB$ is not acceptable since the ext2 maintainers might decide to use these entries for other purposes in the future.

Since there is no direct compatibility between dtfs using an ext2 tradfs implementation and ext2 itself[16]it does not harm compatibility to move around entries in the given ext2 inode data structure. So the 4 byte ext2 inode entry for the atime can be used by dtfs for its own purposes since dtfs stores the atime of the file separately. So the atime entry can be moved around within the ext2 inode structure forming an eight byte entry together with the file size attribute that is already there thus allowing file sizes $> 2GB$.

5.1.3 Benefits for 64 bit Linux platforms

It is already possible to get the VFS layer support large file offsets as it is done by the axp-diffs for Linux/Alpha. On these platforms support for these large files by dtfs should be available without any other changes to either the kernel, the C library or to application programs.

However, 32 bit platforms will not be able to take advantage of dtfs bit file support in the forseeable future.

5.2 Maximum Number Of inodes

Since dtfs stores inodes directly in the .ifile, the maximum file size number possible limits themaximum number of inodes in a file system too. Since the ext2 inode structure is 128 bytes in size and the maximum file size in ext2 is limited to 2 GB, approximately 16 million inodes could be in one file system.

While this number seems to be comfortably large for most cases there might be situations in which this limit is too low. Fortunately, this limit can be overcome by applying the change to the ext2 inode structure described above.

The only problem remaining is the fact that 32 bit platforms cannot pass file sizes $> 2GB$ to user space and have problems with handling 64 bit offsets in general (since the support for the `long long` data type on these platforms provided by gcc seems to be still limited). However, this is not too hard a restriction to live with in the case of the ifile since

- The size of the ifile is not important to user-space programs (as long as there are other means to figure out the maximum number of nodes that is currently allowed, at least for the cleaner).
- dtfs itself accesses the ifile only by using block–addressing and not by using direct file offsets so that the need to deal with offsets that do not fit within a 32 bit word does not arise.

5.3 Delaying Metadata Block Allocations

The current implementation of dtfs makes extensive use of file holes for the meta data information that is mapped to files: After filesystem creation only a small

[16]i.e: you cannot (read-only) mount a dtfs partition as an ext2 partition and vice versa.

fraction (usually only the first block) of the .ifile, the .iusage and the .atime file are actually allocated. However, this causes no problems since the semantic of reading over a hole in a file is well defined as reading an area of nullbytes.

6 Cleaning and Versioning

6.1 A Block-Cleaning Algorithm for Multiple Versions

Minor modifications will be required to the cleaner in order to deal with versioning. The following strategy could be used to perform the live-block test in a filesystem with multiple versions:

1. Perform the live-block test for each filesystem checkpoint. If at least one test with a filesystem checkpoint results in the block in question being marked as live, the block must not be cleaned away.[17]

6.2 Cleaning Partial Segments with Checkpoints

However, problems could arise when the cleaner tries to clean a segment containing a checkpoint. The first issue to deal with in that case is to determine whether the checkpoint is still live.

A checkpoint in a partial segment is considered to be live if it is referenced by *any* checkpoint_entry in *any* checkpoint block or if the segmentt's time stamp is younger than the oldest checkpoint listed in a checkpoint area.[18]

As already mentioned, there is one pair of checkpoint blocks for each logical filesystem.[19] Checkpoint data is alternately written to these two blocks in order to make sure that a valid checkpoint for each logical filesystem version is still present in case of a system failure during a write to a checkpoint block (that could corrupt the data on this block).

Therefore it must be made sure that even the checkpoints referenced by the checkpoint block that is *not* the current one must still be valid.

A simple approach to deal with segments holding live checkpoints when implementing the cleaner would simply be to refuse the cleaning of partial segments containing live checkpoints. Since there are only two checkpoint blocks[20], the total number of active checkpoints for a logical file system will never exceed two times the number of live versions for that logical file system. Since the number of versions for a logical file system will generally be rather small, the total number of live checkpoints will be small, too, so that such a cleaning policy will probably be sufficient.

[17] So having many versions of a logical file system and many different filesystem versions degrades the cleaning performance.

[18] In which case it is a minor checkpoint.

[19] In addition, every checkpoint is replicated once resulting in a total number of four checkpoint blocks for each logical file system.

[20] not counting the replicated versions of the checkpoint blocks

7 Filesystem Versioning

This chapter is an outline of how versioning could be implemented in dtfs by using the existing on-disk data structures. The key data structure for versioning is the dtfs_checkpoint_entry.

7.1 Adding a Version

Creating a new version of a file system can be done by duplicating a checkpoint entry of an existing filesystem version. This filesystem version will then be the predecessor of this newly created version. Therefore it is necessary to specify a predecessor version for every new version to be created.

An implementation of the following algorithm could be used to support versioning within dtfs:

1. Find an empty checkpoint_entry in the checkpoint block for the respective filesystem.[21]

2. Make that empty checkpoint_entry the checkpoint entry for the new version to be created. (Especially set the timestamp to the current timestamp value).

3. Update the timestamp for the checkpoint entry of the current version to the current timestamp value.

4. Write a checkpoint to the next partial segment. Make the checkpoint_entries in the partial segment descriptor point to the entry of the original version and the entry for the new version.

5. Write out the checkpoint block.

Step 5 serves as a commit for all these actions. If the system crashes before step 5 is completed, the filesystem remains in a state that does not have the new version.

7.2 Removing a Version

Removing a version is a rather trivial task:

1. Clear the respective entry in the checkpoint area. (i.e. by setting the timestamp to the NO_TIMESTAMP value.

2. Write out the checkpoint entries.

Reclaiming the additional space used up by a removed version is left to the cleaner.

[21] If none can be found, the maximum number of concurrent versions has already been reached and therefore no more versions can be created for this file system.

References

[SELTZER] Margo Seltzer et. al: An Implementation of a Log-Structured File System for UNIX. Proceedings of the 1993 Winter USENIX Technical Conference

[ROSENBLUM] Mendel Rosenblum: The Design and Implementation of a Log-Structured File System. Diss. 1992

[MERGEMEM] d

[SIPRALOG1] James E. Johnson, William A. Laing: Overview of the Spiralog File System. Digital Technical Journal Volume 8, Issue 2 (Oct. 1996)

[SPIRALOG2] Christopher Whitaker, J. Stuart Bayley, Rod D. W. Widdowson : Design of the Server for the Spiralog File System. Digital Technical Joural Volume 8, Issue 2 (Oct. 1996)

A Changes

$Log: dtfs-disklayout.lyx,v $Revision 1.2 1998/04/01 09:16:54 ceci Updated chapter on versioning (removed version_usage bitmap)

Revision 1.1 1998/03/30 17:45:59 ceciInitial revision